John von Neumann Institute for Computing

Efficient Parallel String Comparison

Peter Krusche and Alexander Tiskin

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 193-200, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Efficient Parallel String Comparison

Peter Krusche and Alexander Tiskin

Department of Computer Science
The University of Warwick, Coventry CV4 7AL, United Kingdom
E-mail: {peter, tiskin}@dcs.warwick.ac.uk[†]

The longest common subsequence (LCS) problem is a classical method of string comparison. Several coarse-grained parallel algorithms for the LCS problem have been proposed in the past. However, none of these algorithms achieve scalable communication. In this paper, we propose the first coarse-grained parallel LCS algorithm with scalable communication. Moreover, the algorithm is work-optimal, synchronisation-efficient, and solves a more general problem of semi-local string comparison, improving in at least two of these aspects on each of the predecessors.

1 Introduction

Computing longest common subsequences of strings is a common method of comparing strings, which is also referred to as string or sequence alignment. It has applications in biology, signal processing and many other areas. Finding the length of the longest common subsequence (LCS) is of interest as a measure of string similarity and is equivalent to finding the Levenshtein string edit distance[9,16]. In the BSP model[15,6], the LCS of two strings of length n can be computed in $O(n^2/p)$ computational work using p processors, $O(n)$ communication and $O(p)$ supersteps using the standard dynamic programming algorithm[16] combined with the grid dag method[10] (see also[2,7]). In addition, various algorithmic applications[12] require computing the LCS lengths for a string against all substrings of the other string, and/or the LCS lengths for all prefixes of one string against all suffixes of the other string. These additional tasks can be performed in the BSP model at no extra asymptotic cost by combining the grid dag method with the sequential algorithm by Alves et al.[4] or the one by Tiskin[14] (both based on an algorithm by Schmidt[11]).

Alternative algorithms by Tiskin[13,14] compute LCS lengths using a fast method for $(\max, +)$ multiplication of highest-score matrices in an implicit (critical point) representation. Using this method, two highest-score matrices with n critical points each can be multiplied in time $O(n^{1.5})$. Overall, parallel computation of the implicit highest-score matrix for two given strings can be performed in local computation $\mathsf{W} = O(n^2/p)$, communication $\mathsf{H} = O(n \log p)$ and $\mathsf{S} = \log p$ supersteps.

None of the described algorithms achieve scalable communication, i.e. communication $O(n/p^\alpha)$ with $\alpha > 0$. In this paper, we propose the first BSP algorithm with scalable communication, running in local computation $\mathsf{W} = O(n^2/p)$, communication $\mathsf{H} = O(n \log p/\sqrt{p})$ and $\mathsf{S} = \log p$ supersteps. The key idea of the algorithm is to carry out the highest-score matrix multiplication procedure in parallel and in a constant number of supersteps. An overview of previous results and new improvements is given in Table 1.

[†]The authors acknowledge the support of DIMAP (the Centre for Discrete Mathematics and its Applications) during this work.

Table 1. Parallel algorithms for LCS/Levenshtein distance computation

global / str.-substr. / prefix-suffix	W	H	S	*References*
• / - / -	$O(\frac{n^2}{p})$	$O(n)$	$O(p)$	[10]+[16]
• / • / •	$O(\frac{n^2}{p})$	$O(n)$	$O(p)$	[10]+[4,14]
• / • / •	$O(\frac{n^2 \log n}{p})$	$O(\frac{n^2 \log p}{p})$	$O(\log p)$	[1]
• / • / -	$O(\frac{n^2}{p})$	$O(pn \log p)$	$O(\log p)$	[3]
• / • / -	$O(\frac{n^2}{p})$	$O(n \log p)$	$O(\log p)$	[13,4]
• / • / •	$O(\frac{n^2}{p})$	$O(\frac{n \log p}{\sqrt{p}})$	$O(\log p)$	NEW

2 The BSP Model

The BSP model introduced by Valiant[15] describes a parallel computer with three parameters (p, g, l). The performance of the communication network is characterised by a linear approximation, using parameters g and l. Parameter g, the *communication gap*, describes how fast data can be transmitted continuously by the network (in relation to the computation speed of the individual processors) after the transfer has started. The *communication latency* l represents the overhead that is necessary for starting up communication. A BSP computation is divided into *supersteps*, each consisting of local computations and a communication phase. At the end of each superstep, the processes are synchronised using a barrier-style synchronisation. Consider a computation consisting of S supersteps. For each specific superstep $1 \leq \mathsf{s} \leq \mathsf{S}$ and each processor $1 \leq q \leq p$, let $\mathsf{h}^{in}_{\mathsf{s},q}$ be the maximum number of data units received and $\mathsf{h}^{out}_{\mathsf{s},q}$ the maximum number of data units sent in the communication phase on processor q. Further, let w_s be the maximum number of operations in the local computation phase. The whole computation has separate *computation cost* $\mathsf{W} = \sum_{\mathsf{s}=1}^{\mathsf{S}} \mathsf{w}_\mathsf{s}$ and *communication cost* $\mathsf{H} = \sum_{\mathsf{s}=1}^{\mathsf{S}} \mathsf{h}_{\mathsf{s},q}$ with $\mathsf{h}_{\mathsf{s},q} = \max_{1 \leq q \leq p}(\mathsf{h}^{in}_{\mathsf{s},q} + \mathsf{h}^{out}_{\mathsf{s},q})$. The total running time is given by the sum $\mathsf{T} = \sum_{\mathsf{s}=1}^{\mathsf{S}} \mathsf{T}_\mathsf{s} = \mathsf{W} + g \cdot \mathsf{H} + l \cdot \mathsf{S}$.

3 Problem Analysis and Sequential Algorithm

Let $x = x_1 x_2 \ldots x_m$ and $y = y_1 y_2 \ldots y_n$ be two strings over an alphabet Σ. A *substring* of any string x can be obtained by removing zero or more characters from the beginning and/or the end of x. A *subsequence* of string x is any string that can be obtained by deleting zero or more characters, i.e. a string with characters x_{j_k}, where $j_k < j_{k+1}$ and $1 \leq j_k \leq m$ for all k. The *longest common subsequence* of two strings is the longest string that is a subsequence of both input strings. A common approach to finding the LCS of two strings is to define a grid directed acyclic graph, which has vertical and horizontal edges of weight 0, and diagonal edges of weight 1 for every character match between the input strings. Let this *alignment dag* be defined by a set of vertices $v_{i,j}$ with $i \in \{0, 1, 2. \ldots, m\}$ and $j \in \{0, 1, 2. \ldots, n\}$ and edges as follows. We have horizontal and vertical edges $v_{i,j-1} \rightarrow v_{i,j}$ and $v_{i-1,j} \rightarrow v_{i,j}$ of weight 0, and diagonal edges $v_{i-1,j-1} \rightarrow v_{i,j}$ of

weight 1 that are present only when $x_i = y_j$. Longest common subsequences of a substring $x_i x_{i+1} \dots x_j$ and y correspond to longest paths in this graph from $v_{i-1,0}$ to $v_{j,m}$. In addition to the standard LCS problem in which only strings x and y are compared, we consider its generalisation *semi-local LCS*, which includes computation of the lengths of the string-substring LCS, the prefix-suffix LCS and symmetrically the substring-string LCS and the suffix-prefix LCS. Solutions to the semi-local LCS problem can be represented by a matrix $A(i, j)$, where each entry is related to the length of the longest common subsequence of y and substring $x_i \dots x_j$ (or of substrings $x_1 \dots x_i$ and $y_j \dots y_n$, etc.). Without loss of generality, we will assume from now on that both input strings have the same length n. In this case, matrix A has size $N \times N$ with $N = 2n$.

The sequential algorithm by Tiskin[14] computes a set of N *critical points* that can be used to query the length of the LCS of string y and any substring $x_i \dots x_j$. These critical points are given as (odd) half-integer pairs $(\hat{\imath}, \hat{\jmath})$ (corresponding to positions between the vertices of the alignment dag). Throughout this paper we will denote half-integer variables using a $\hat{}$, and denote the set of half-integers $\{i+\frac{1}{2}, i+\frac{3}{2}, \dots, j-\frac{1}{2}\}$ as $\langle i : j \rangle$ (analogously, we denote the set of integers $\{i, i+1, \dots, j\}$ as $[i : j]$). A point (r_1, c_1) is *dominated* by another point (r_2, c_2) if $r_1 \geq r_2$ and $c_1 \leq c_2$. It has been shown[14] that there are N critical points, such that every entry $A(i, j)$ can be computed as $A(i, j) = j - i - a(i, j)$, where $a(i, j)$ is the number of critical points that are dominated by the pair of integers (i, j). This set of critical points defines a permutation matrix D_A, which is used as an implicit representation of the highest-score matrix A. Having computed all critical points, querying the values for $A(i, j)$ is possible in $O(\log^2 N)$ time per query by using a range tree[5], or by using an asymptotically more efficient data structure[8].

Furthermore, Tiskin[14] describes a procedure which, given two highest-score matrices that correspond to adjacent strips in the grid dag, computes the highest-score matrix for the union of these two strips. This procedure can be reduced to the multiplication of two integer matrices in the $(\min, +)$ semiring. As an input, we have the nonzeros of two $N \times N$ permutation matrices D_A and D_B. The procedure computes an $N \times N$ permutation matrix D_C, such that the permutation distribution matrix d_C is the $(\min, +)$ product of permutation distribution matrices d_A and d_B. The permutation distribution matrix d_C is defined as

$$d_C(i, k) = \sum_{(\hat{\imath}, \hat{k}) \in \langle i:N \rangle \times \langle 0:k \rangle} D_C(\hat{\imath}, \hat{k}), \tag{3.1}$$

and d_A and d_B are defined analogously. Matrices D_A, D_B and D_C have half-integer indices ranging over $\langle 0 : N \rangle$. Furthermore, we assume without loss of generality that N is a power of 2. By exploiting the monotonicity properties of permutation-distribution matrices, the procedure runs in time $O(N^{1.5})$ for matrices of size $N \times N$, given by their implicit (critical point) representation. The method uses a divide-and-conquer approach that recursively partitions the output matrix into smaller blocks in order to locate its nonzeros.

We will now describe the matrix multiplication method in more detail. As an input, we have the nonzeros of the permutation matrices D_A and D_B, which are both of size $N \times N$. The procedure computes the permutation matrix D_C. At every level of the recursion, we consider a square block in D_C corresponding to the set of indices $\langle i_0 - h : i_0 \rangle \times \langle k_0 : k_0 + h \rangle$. We will call such a block a *C-block*, and denote every such block by the triple (i_0, k_0, h). For every C-block, we compute the number of nonzeros contained within. We

can stop recursing in two cases: if the C-block does not contain any nonzeros, or if it is of size 1×1 and thus specifies the location of a nonzero. The number of nonzeros in a C-block is computed as follows. For each C-block, we define a subset of *relevant* nonzeros in D_A with indices $\mathcal{I}^{(i_0,k_0,h)} = \{(\hat{\imath},\hat{\jmath}) \in \langle i_0 - h : i_0\rangle \times \langle 0 : N\rangle$ and $D_A(\hat{\imath},\hat{\jmath}) = 1\}$ and a subset of relevant nonzeros in D_B with indices $\mathcal{K}^{(i_0,k_0,h)} = \{(\hat{\jmath},\hat{k}) \in \langle 0 : N\rangle \times \langle k_0 : k_0 + h\rangle$ and $D_B(\hat{\jmath},\hat{k}) = 1\}$. We can split a given set of relevant nonzeros in D_A and D_B into two sets at a position $j \in [0 : N]$, and determine the numbers of relevant nonzeros in D_A up to and including column $j - \frac{1}{2}$:

$$\delta_A^{(i_0,k_0,h)}(j) = \mid \{(\hat{\imath},\hat{\jmath}) \in \mathcal{I}^{(i_0,k_0,h)} \text{ and } \hat{\jmath} < j\} \mid = d_A(i_0 - h, j) - d_A(i_0, j), \tag{3.2}$$

as well as the number of relevant nonzeros in D_B starting at row $j + \frac{1}{2}$:

$$\delta_B^{(i_0,k_0,h)}(j) = \mid \{(\hat{\jmath},\hat{k}) \in \mathcal{K}^{(i_0,k_0,h)} \text{ and } \hat{\jmath} > j\} \mid = d_B(j, k_0 + h) - d_B(j, k_0). \tag{3.3}$$

These sequences can be obtained in time $O(N)$ by a scan of the relevant nonzeros in D_A and D_B, which are given as input[a][b]. At lower levels of the recursion, it is possible to compute the values from the sequences that were computed at the previous level. Since, for a fixed C-block, δ_A and δ_B only change at the values of j at which also the number of nonzeros in the relevant part of D_A or D_B changes, we can define contiguous sets of j, which we call the "*j-blocks*", corresponding to a value of $d \in [-h : h]$ which uniquely identifies this block. We define $\mathcal{J}^{(i_0,k_0,h)}(d) = \{j \mid \delta_A^{(i_0,k_0,h)}(j) - \delta_B^{(i_0,k_0,h)}(j) = d\}$. When $\mathcal{I}^{(i_0,k_0,h)}$ and $\mathcal{K}^{(i_0,k_0,h)}$ are given, we can determine the j-blocks by an $O(h)$ scan[c] of these sets. Notice that a j-block need not exist for every d. Particularly for small C-blocks, there will be few j-blocks, as the number of relevant nonzeros decreases with the block size. On the set of existing j-blocks, we define sequences

$$\Delta_A^{(i_0,k_0,h)}(d) = \operatorname*{any}_j \delta_A^{(i_0,k_0,h)}(j) \quad \text{and} \quad \Delta_B^{(i_0,k_0,h)}(d) = \operatorname*{any}_j \delta_B^{(i_0,k_0,h)}(j). \tag{3.4}$$

The predicate "any" is taken over all j in the j-block $\mathcal{J}^{(i_0,k_0,h)}(d)$ corresponding to d. This is sufficient since for a fixed value of d, the corresponding values of $\delta_A(j)$ and $\delta_B(j)$ are equal for all j in the j-block defined by d. When the value of either δ_A or δ_B changes, the corresponding value of d changes, too. For each C-block, we are also interested in the sequence of minima

$$M^{(i_0,k_0,h)}(d) = \min_j \left(d_A(i_0, j) + d_B(j, k_0)\right), \text{ with } j \in \mathcal{J}^{(i_0,k_0,h)}(d). \tag{3.5}$$

The predicate "min" is taken over all j in the j-blocks corresponding to d since the values $d_A(i_0, j) + d_B(j, k_0)$ can be different inside a j-block.

At the top level, the set $\mathcal{J}^{(N,0,N)}(d)$ always contains either one or zero elements, and the corresponding sequence $M^{(N,0,N)}(d) = 0$ for all d. At lower levels of the recursion

[a] If the nonzeros are given as tuples, pre-sorting all tuples $(\hat{\imath},\hat{\jmath})$ from D_A and $(\hat{\jmath},\hat{k})$ from D_B by $\hat{\jmath}$ can be used to simplify the scanning procedure as this allows to access the tuples $(\hat{\imath},\hat{\jmath})$ in $\mathcal{I}^{(i_0,k_0,h)}$ and accordingly $(\hat{\jmath},\hat{k})$ in $\mathcal{K}^{(i_0,k_0,h)}$ in order of $\hat{\jmath}$. The order can be preserved through the different levels of the recursion, and the sorting can be carried out by bucket sorting without increasing the memory or computation cost.

[b] If the whole matrix D_C is used as a block for starting the recursion, these sequences are trivial, having $\delta_A^{(N,0,N)}(j) = j$ and $\delta_B^{(N,0,N)}(j) = N - j$, as $\mathcal{I}_{(N,0,N)}$ contains all nonzeros in the permutation matrix D_A and $\mathcal{K}_{(N,0,N)}$ contains all nonzeros in the permutation matrix D_B.

[c] We have $\mid \mathcal{I}^{(i_0,k_0,h)} \mid = \mid \mathcal{K}^{(i_0,k_0,h)} \mid = h$ because D_A and D_B are permutation matrices.

tree (i.e. for smaller block sizes h), the number of relevant nonzeros in D_A and D_B decreases, and the individual j-blocks contain more elements. At any level of the recursion tree, sequence $M^{(i_0,k_0,h)}$ can be computed by an $O(N)$ scan of the sets $\mathcal{I}^{(i_0,k_0,h)}$ and $\mathcal{K}^{(i_0,k_0,h)}$. However, at lower levels of the recursion we can also determine the sequence M corresponding to any C-subblock of any C-block (i_0, k_0, h) using an $O(h)$ scan of sequences $M^{(i_0,k_0,h)}$, $\Delta_A^{(i_0,k_0,h)}$ and $\Delta_B^{(i_0,k_0,h)}$. Using sequences $M^{(i_0,k_0,h)}$, $\Delta_A^{(i_0,k_0,h)}$ and $\Delta_B^{(i_0,k_0,h)}$, it is possible to obtain the values of d_C at the four corners of the current C-block in time $O(h)$ by taking the minimum over all values $d \in [-h : h]$ for which a j-block exists:

$$\begin{aligned} d_C(i_0, k_0) &= \min_d M^{(i_0,k_0,h)}(d), \\ d_C(i_0 - h, k_0) &= \min_d (\Delta_A^{(i_0,k_0,h)}(d) + M^{(i_0,k_0,h)}(d)), \\ d_C(i_0, k_0 + h) &= \min_d (\Delta_B^{(i_0,k_0,h)}(d) + M^{(i_0,k_0,h)}(d)), \\ d_C(i_0 - h, k_0 + h) &= \min_d (\Delta_A^{(i_0,k_0,h)}(d) + \Delta_B^{(i_0,k_0,h)}(d) + M^{(i_0,k_0,h)}(d)). \end{aligned} \tag{3.6}$$

From these values, the number of nonzeros in the current block can be obtained by computing $d_C(i_0 - h, k_0 + h) - d_C(i_0 - h, k_0) - d_C(i_0, k_0 + h) + d_C(i_0, k_0)$. If this number is zero, the recursion can terminate at the current C-block. Otherwise, the algorithm proceeds to recursively partition the block into four subblocks of size $\frac{h}{2}$ in order to locate the nonzeros. In order to partition the block, it is necessary to determine the sequences Δ_A, Δ_B and M for all four C-subblocks $(i', k', \frac{h}{2})$ with $(i', k') \in \{i_0, i_0 - \frac{h}{2}\} \times \{k_0, k_0 + \frac{h}{2}\}$. To determine the sequences Δ_A and Δ_B, it is sufficient to scan the $\frac{h}{2}$ relevant nonzeros in every block. To establish the sequence M for every C-subblock for all $d' \in [-\frac{h}{2} : \frac{h}{2}]$, we define sequences $\bar{\Delta}$ that contain the number of relevant nonzeros in subblock $(i', k', \frac{h}{2})$ for every j-block $\mathcal{J}(d)$ of the current C-block as

$$\bar{\Delta}_A^{(i',k',\frac{h}{2})}(d) = \operatorname*{any}_{j \in \mathcal{J}(d)} \delta_A^{(i',k',\frac{h}{2})}(j) \quad \text{and} \quad \bar{\Delta}_B^{(i',k',\frac{h}{2})}(d) = \operatorname*{any}_{j \in \mathcal{J}(d)} \delta_B^{(i',k',\frac{h}{2})}(j). \tag{3.7}$$

Using these sequences, it is possible to compute the sequences M for every C-subblock from only the minima corresponding to the current C-block and the relevant nonzeros in the C-subblock by taking

$$\begin{aligned} M^{(i_0,k_0,\frac{h}{2})}(d') &= \min_d M^{(i_0,k_0,h)}(d), \\ M^{(i_0,k_0+\frac{h}{2},\frac{h}{2})}(d') &= \min_d M^{(i_0,k_0,h)}(d) + \bar{\Delta}_B^{(i_0,k_0+\frac{h}{2},\frac{h}{2})}(d), \\ M^{(i_0-\frac{h}{2},k_0,\frac{h}{2})}(d') &= \min_d M^{(i_0,k_0,h)}(d) + \bar{\Delta}_A^{(i_0-\frac{h}{2},k_0,\frac{h}{2})}(d), \\ M^{(i_0-\frac{h}{2},k_0+\frac{h}{2},\frac{h}{2})}(d') &= \min_d M^{(i_0,k_0,h)}(d) + \bar{\Delta}_A^{(i_0-\frac{h}{2},k_0+\frac{h}{2},\frac{h}{2})}(d) + \bar{\Delta}_B^{(i_0-\frac{h}{2},k_0+\frac{h}{2},\frac{h}{2})}(d) \end{aligned} \tag{3.8}$$

over all d such that $\bar{\Delta}_A^{(i',k',\frac{h}{2})}(d) - \bar{\Delta}_B^{(i',k',\frac{h}{2})}(d) = d'$ with $d' \in [-\frac{h}{2} : \frac{h}{2}]$. This is equivalent to computing the j-blocks for the C-subblocks. Notice the difference in index between sequences Δ and $\bar{\Delta}$: $\bar{\Delta}$ counts the numbers of relevant nonzeros corresponding to the current C-block, whereas the sequences Δ count numbers of relevant nonzeros in the C-subblocks of the current C-block.

4 Parallel Algorithm

The sequential highest-score matrix multiplication procedure can be used to derive a parallel algorithm[13] that solves the semi-local LCS problem by partitioning the alignment dag into p strips. The problem is solved independently on one processor for each strip using dynamic programming[11,4] to compute the implicit highest-score matrices, and then "merging" the resulting highest-score matrices in a binary tree of height $\log p$. This procedure requires data of size $O(N)$ to be sent by every processor in every level of the tree. The sequential merging requires time $O(N^{1.5})$ for computing the resulting highest-score matrix.

By parallelising the highest-score matrix multiplication algorithm and partitioning the alignment dag into a grid of $\sqrt{p} \times \sqrt{p}$ square blocks, we reduce the number of critical points that need to be transferred in every level of the tree to $O(N/\sqrt{p})$ per processor. We assume w.l.o.g. that $\sqrt{p}$ is an integer, and that every processor has an unique identifier q with $0 \leq q < p$. Furthermore, we assume that every processor q corresponds to exactly one pair $(q_x, q_y) \in [0 : \sqrt{p} - 1] \times [0 : \sqrt{p} - 1]$.

We now describe the parallel version of the highest-score matrix multiplication algorithm. The initial distribution of the nonzeros of the input matrices is assumed to be even among all processors, so that every processor holds $\frac{N}{p}$ nonzeros of D_A and D_B. The recursive divide-and-conquer computation from the previous section has at most p independent problems at level $\frac{1}{2}\log_2 p$. In the parallel version of the algorithm, we start the recursion directly at this level, computing relevant nonzeros and sequence M from scratch as follows.

First we redistribute the nonzeros to strips of width $\frac{N}{p}$ by sending each nonzero $(\hat{\imath}, \hat{\jmath})$ in D_A and each nonzero $(\hat{\jmath}, \hat{k})$ in D_B to processor $\lfloor(\hat{\jmath} - \frac{1}{2}) \cdot p/N\rfloor$. This is possible in one superstep using communication $O(\frac{N}{p})$. To compute the values of sequence M for every processor (M-values), we compute the elementary $(\min, +)$ products $d_A(q_x \cdot \frac{N}{\sqrt{p}}, j) + d_B(j, q_y \cdot \frac{N}{\sqrt{p}})$ for all $j \in [0 : N]$ and every pair (q_x, q_y). Every processor holds all $D_A(\hat{\imath}, \hat{\jmath})$ and all $D_B(\hat{\jmath}, \hat{k})$ for $\hat{\jmath} \in \langle q \cdot \frac{N}{\sqrt{p}} : (q+1) \cdot \frac{N}{\sqrt{p}}\rangle$. Since d_A and d_B are defined from D_A and D_B by (3.1), we can compute the values $d_A(q_x \cdot \frac{N}{\sqrt{p}}, j)$ and $d_B(j, q_y \cdot \frac{N}{\sqrt{p}})$ in blocks of $\frac{N}{p}$ on every processor by using a parallel prefix (respectively parallel suffix) operation. We have $\sqrt{p}$ instances of parallel prefix (respectively parallel suffix), one for each value of q_x (respectively q_y). Therefore, the total cost of the parallel prefix and suffix computation is $\mathsf{W} = O(\frac{N}{p} \cdot \sqrt{p}) = O(\frac{N}{\sqrt{p}})$; the communication cost is negligible as long as $\frac{N}{p} \geq p \;\Rightarrow N \geq p^2$. The parallel prefix and suffix operations can be carried out in $\mathsf{S} = O(1)$ supersteps by computing intermediate (local) prefix results on every processor, performing an all-to-all exchange of these values, and then locally combining on every processor the local results with the corresponding intermediate values. After the prefix and suffix computations, every processor holds N/p values $d_A(q_x \cdot \frac{N}{\sqrt{p}}, j) + d_B(j, q_y \cdot \frac{N}{\sqrt{p}})$ for $j \in [q \cdot \frac{N}{\sqrt{p}} : (q+1) \cdot \frac{N}{\sqrt{p}}]$.

Now we redistribute the data, assigning a different C-block $(q_x \cdot \frac{N}{\sqrt{p}}, q_y \cdot \frac{N}{\sqrt{p}}, \frac{N}{\sqrt{p}}, \frac{N}{\sqrt{p}})$ to each processor q. To be able to continue the recursive procedure at this level, every processor must have the $O(\frac{N}{\sqrt{p}})$ values of sequence $M^{(q_x \cdot \frac{N}{\sqrt{p}}, q_y \cdot \frac{N}{\sqrt{p}}, \frac{N}{\sqrt{p}}, \frac{N}{\sqrt{p}})}$, and the sets of relevant

nonzeros in D_A and D_B. Each processor holds at most N/p nonzeros in D_A and the same number of nonzeros in D_B. Imagine that the nonzeros are added one by one to initially empty matrices D_A and D_B. Each nonzero in D_A (respectively in D_B) can increase the overall number of j-blocks by at most 1 for each of the $\sqrt{p}$ C-blocks where this nonzero is relevant; a nonzero does not affect the number of j-blocks for any other C-blocks. Each j-block is assigned one value in the corresponding sequence M. Therefore, the total number of M-values per processor before redistribution is at most $N/p \cdot \sqrt{p} + p = N/\sqrt{p} + p$. The total number of M-values per processor after redistribution is $N/\sqrt{p}$, therefore the communication is perfectly balanced, apart from the maximum of p values that can arise due to processor boundaries "splitting" a j-block. Since every processor holds N/p nonzeros before redistribution, and every nonzero is relevant for $\sqrt{p}$ C-blocks, redistributing the relevant nonzeros can also be done in $O(N/\sqrt{p})$ communication. After this, every processor has all the data that are necessary to perform the sequential procedure from the previous section on its C-block. The resulting parallel highest-score matrix multiplication procedure has BSP cost $\mathsf{W} = O((N/\sqrt{p})^{1.5}) = O(N^{1.5}/p^{0.75})$, $\mathsf{H} = O(N/\sqrt{p})$ and $\mathsf{S} = O(1)$.

When applied to semi-local LCS computation, this algorithm is used at every level of a quadtree merging scheme. At the bottom level, the alignment dag is partitioned into a regular grid of p sub-dags of size $n/\sqrt{p} \times n/\sqrt{p}$. The highest-score matrix for each sub-dag is computed sequentially by a separate processor in computation work $O(N^2/p)$. Then the matrices are merged sequentially with computation work $O((N/\sqrt{p})^{1.5}) = O(N^{1.5}/p^{0.75})$. At higher levels of the quadtree, blocks are merged in parallel. In particular at level $\log r$, $1 \leq r \leq p$, the block size is $N/\sqrt{r}$, and each merge is performed by a group of p/r processors in computation work $\mathsf{W} = O(\frac{(N/r^{0.5})^{1.5}}{(p/r)^{0.75}}) = O(\frac{N^{1.5}/r^{0.75}}{p^{0.75}/r^{0.75}}) = O(\frac{n^{1.5}}{p^{0.75}})$ and communication $\mathsf{H} = O(\frac{(n/r^{0.5})}{(p/r)^{0.5}}) = O(\frac{n}{p^{0.5}})$. This analysis includes the root of the quadtree, where $r = 1$. Overall, the new algorithm runs in local computation $\mathsf{W} = O(\frac{n^2}{p} + \frac{N^{1.5}\log p}{p^{0.75}}) = O(n^2/p)$ (assuming that $n \geq p^2$), communication $\mathsf{H} = O(\frac{n \log p}{\sqrt{p}})$ and $\mathsf{S} = O(\log p)$ supersteps.

5 Conclusions and Outlook

In this paper, we have presented an efficient coarse grained parallel algorithm for semi-local string comparison based on a parallel highest-score matrix multiplication procedure. Our algorithm reduces the BSP overall communication cost H to $O(n \log p/\sqrt{p})$, running in local computation $\mathsf{W} = O(n^2/p)$ and using $\mathsf{S} = O(\log p)$ supersteps. The local computation cost can be reduced slightly by using a subquadratic sequential algorithm[14]. Thus, our algorithm is the first coarse-grained parallel LCS algorithm with scalable communication. Moreover, the algorithm is work-optimal, synchronisation-efficient, and solves a more general problem of semi-local string comparison. It is worth mentioning here that the algorithm can be extended to allow querying the actual longest common subsequences (as opposed to just their lengths). Also, the data structures used by this algorithm allow compact storage of the results, which is of advantage when comparing very large strings. Since this algorithm is a useful building block for solving various algorithmic problems[12], we intend to implement this algorithm in order to investigate its practicality and to provide an "algorithmic plug-in" for these applications.

References

1. C. E. R. Alves, E. N. Cáceres, F. Dehne and S. W. Song, *Parallel dynamic programming for solving the string editing problem on a CGM/BSP*, in: Proc. of 14th ACM SPAA, pp. 275–281, (2002).
2. C. E. R. Alves, E. N. Cáceres, F. Dehne and S. W. Song, *A parallel wavefront algorithm for efficient biological sequence comparison*, in: Proc. ICCSA, vol. **2668** of LNCS, (2003).
3. C. E. R. Alves, E. N. Cáceres and S. W. Song, *A BSP/CGM algorithm for the all-substrings longest common subsequence problem*, in: Proc. 17th IEEE/ACM IPDPS, pp 1–8, (2003).
4. C. E. R. Alves, E. N. Cáceres and S. W. Song, *A coarse-grained parallel algorithm for the all-substrings longest common subsequence problem*, Algorithmica, **45**, 301–335, (2006).
5. J. L. Bentley, *Multidimensional divide-and-conquer*, Comm. ACM, **23**, 214–229, (1980).
6. R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*, (Oxford University Press, 2004).
7. T. Garcia and D. Semé, *A load balancing technique for some coarse-grained multicomputer algorithms*, in: SCA 21st International Conference on Computers and Their Applications (CATA '06), pp. 301–306, (2006).
8. J. Jájá, C. Mortensen and Q. Shi, *Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Range Counting*, in: Proc. 15th ISAAC, vol. **3341** of LNCS, (2004).
9. V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions and reversals*, Sov. Phys. Dokl., **6**, 707–710, (1966).
10. W. F. McColl, *Scalable Computing*, in: Computer Science Today: Recent Trends and Developments, vol. **1000** of LNCS, pp. 46–61, (Springer-Verlag, 1995).
11. J. P. Schmidt, *All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings*, SIAM Journal on Computing, **27**, 972–992, (1998).
12. A. Tiskin, *Semi-local string comparison: Algorithmic techniques and applications*, Mathematics in Computer Science, `arXiv:0707.3619 [cs.DS]`, (2007, in press).
13. A. Tiskin. *Efficient representation and parallel computation of string-substring longest common subsequences*, in: Proc. ParCo05, vol. **33** of NIC Series, pp. 827–834, (John von Neumann Institute for Computing, 2005).
14. A. Tiskin, *All semi-local longest common subsequences in subquadratic time*, *Journal of Discrete Algorithms*, (2008, in press).
15. L. G. Valiant, *A bridging model for parallel computation*, Comm. ACM, **33**, 103–111, (1990).
16. R. A. Wagner and M. J. Fischer, *The string-to-string correction problem*, J. ACM, **21**, 168–173, (1974).